
Please answer each of the following problems. Refer to the course webpage for the **collaboration policy**, as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

1. Suppose that p is an unknown value, $0 < p < 1$. Suppose that you can call a function **randP** which returns **true** with probability p and returns **false** with probability $1 - p$. Every call to **randP** is independent. You have no way to generate random numbers except through **randP**.
 - (a) (2 pts) Describe an algorithm—using **randP**—that returns **true** with probability $1/2$ and **false** with probability $1/2$. Your algorithm should in expectation, use $\frac{1}{p(1-p)}$ calls to **randP**. Your algorithm does not have access to the value of p , and does not have access to any source of randomness other than calls to **randP**. **[We are expecting pseudocode, and a short English description of what the algorithm does. Your pseudocode should be detailed enough that a CS106B student could implement it without much thought.]**
Hints: (i) Your algorithm does not have to compute p , or an approximation to it. (ii) Notice that in the worst case, your algorithm may use more calls to **randP**, possibly even infinitely many.
 - (b) (1pts) Formally prove that your algorithm runs using expected $\frac{1}{p(1-p)}$ calls to **randP**. **[We are expecting a mathematical calculation of the expected value of the total number of calls to randP.]**
 - (c) (1 pt) Informally argue that your algorithm returns **true** with probability $1/2$ and **false** with probability $1/2$. **[We are expecting an informal justification of why the algorithm returns true with probability $1/2$ and false with probability $1/2$. Your argument should convince the reader that you are correct, but does not have to be a formal proof.]**

SOLUTION.

- (a) The pseudocode is as follows.

```
while True:
    x1 = randP()
    x2 = randP()
    if x1 != x2:
        return x1
```

That is, the algorithm draws two samples from **randP**. If they are not equal, it returns the first one. Intuitively, no matter what p is, if $x_1 \neq x_2$, then half the time it should be because $x_1 = 0$ and $x_2 = 1$, and half the time it should be the other way around.

- (b) We analyze the runtime of the algorithm above. We actually give three different solutions: any one of them is acceptable.

Solution 1. Let X be the number of iterations required. We want to find $E[X]$.

Let A be the event that the algorithm terminates after the first iteration, and let the complement, $\bar{A}$, be the event that it does not. Recall that the law of total expectation states that

$$E[X] = E[X|A]P(A) + E[X|\bar{A}]P(\bar{A}).$$

In this case, we have $E[X|A] = 1$ (that is, the number of iterations is 1 if the algorithm terminates after the first iteration), and the probability that A occurs is $P(A) = 2p(1-p)$; this is the probability that we draw either $x_1 = 0$ and $x_2 = 1$ or $x_1 = 1$ and $x_2 = 0$. On the other hand, $E[X|\bar{A}] = 1 + E[X]$. This is because if the algorithm does not terminate after the first step, it is back where it started. Putting this all together, we have:

$$E[X] = 1 \cdot 2p(1-p) + (E[X] + 1) \cdot (1 - 2p(1-p))$$

Solving for $E[X]$, we get

$$2p(1-p)E[X] = 2p(1-p) - 2p(1-p) + 1$$

and

$$E[X] = \frac{1}{2p(1-p)}.$$

Therefore, the expected number of coin flips required is $\frac{1}{p(1-p)}$.

Solution 2. For each iteration, the probability that $x_1 \neq x_2$ is $2p(1-p)$. For each integer $k > 0$, the probability that we need exactly k iterations is $(1 - 2p(1-p))^{k-1} \cdot 2p(1-p)$. Thus the average number of iterations is

$$X = \sum_{k=1}^{\infty} k(1 - 2p(1-p))^{k-1} \cdot 2p(1-p) = 2p(1-p) \cdot \sum_{k=1}^{\infty} k(1 - 2p(1-p))^{k-1}$$

Denote $S = \sum_{k=1}^{\infty} kc^{k-1}$ where $c = (1 - 2p(1-p))$. Then $cS = \sum_{k=1}^{\infty} kc^k$ and thus

$$(1-c)S = S - cS = (1 + 2c + 3c^2 + \dots) - (c + 2c^2 + \dots) = \sum_{k=0}^{\infty} c^k = \frac{1}{1-c}$$

So

$$S = \frac{1}{(1-c)^2} = \frac{1}{(2p(1-p))^2}$$

and

$$X = 2p(1-p) \cdot S = \frac{2p(1-p)}{(2p(1-p))^2} = \frac{1}{2p(1-p)}$$

and since there are two coin flips per iteration, the expected number of coin flips is

$$\frac{1}{p(1-p)}$$

Solution 3. For each iteration, the probability that the algorithm terminates after that iteration is $2p(1 - p)$. The number of iterations required for the procedure to terminate follows a geometric distribution with parameter $2p(1 - p)$. Thus, the expected number of iterations is

$$\frac{1}{2p(1 - p)}$$

and the expected number of coin flips is

$$\frac{1}{p(1 - p)}$$

- (c) Finally, we analyze the correctness. The probability that the algorithm returns **true** on iteration i is the same as the probability that it returns **false** on iteration i . This is because the algorithm returns **true** if $x_1 = \text{true}$ and $x_2 = \text{false}$, which happens with probability $p(1 - p)$. It returns **false** if $x_2 = \text{true}$ and $x_1 = \text{false}$, which also happens with probability $p(1 - p)$.

At this point, for full credit it is enough to say that since whenever the algorithm returns, it has a $1/2$ chance of returning 0 or 1, then overall it has a $1/2$ chance of returning 0 or 1. Formally, we may proceed as follows:

Let B_i be the event that the algorithm returns on iteration i . Then by symmetry

$$P(\text{return false}|B_i) = P(\text{return true}|B_i).$$

Thus, using the law of total probability again,

$$P(\text{return true}) = \sum_{i=1}^{\infty} P(\text{return true}|B_i)P(B_i) = \sum_{i=1}^{\infty} P(\text{return false}|B_i)P(B_i) = P(\text{return false}).$$

2. Suppose that A and B are sorted arrays of length n , and that all numbers in the arrays are distinct.
- (a) (3pts) Design an algorithm to find the median of all $2n$ numbers in $O(\log n)$ time. For our purposes, we define the median of the $2n$ numbers as the n th smallest number in the $2n$ values. **[We are expecting: pseudocode, and an English description of the algorithm, detailed enough that a CS 106B student could implement it without much thought.]**
 - (b) (3pts) Informally argue that your algorithm correctly finds the median of all $2n$ numbers. **[We are expecting a short (paragraph or two) argument that will convince the reader why your algorithm works correctly.]**
 - (c) (2pts) Prove that your algorithm runs in time $O(\log(n))$ time. **[We are expecting a formal proof.]**

SOLUTION:

The idea is to discard an equal number of elements above and below the median, and recurse on two smaller arrays, maintaining the invariant that the median of the elements in the smaller arrays equals the true median of the $2n$ elements. The correctness proof relies on proving this invariant holds.

- (a) **English description:** Compare the medians of the two arrays. Recurse on the upper half of the array with the smaller median, and the lower half of the array with the larger median. When the subarrays you are considering have size 1, simply return the minimum of the two elements, as a base case.

In the pseudocode below, we choose to cut the array in half when the length is even, and simply shorten it by one when the length is odd, so as to make it even again. There are other correct ways (for example, using floors and ceilings to cut the array approximately in half each time).

```
def median_helper(A, B, a_low, a_high, b_low, b_high):
    a_mid = (a_low + a_high) / 2
    b_mid = (b_low + b_high) / 2

    if (a_low == a_high):
        return min(A[a_low], B[b_low])
    elif (a_high - a_low) % 2 == 0:
        if A[a_mid] < B[b_mid]:
            return median_helper(A, B, a_mid, a_high, b_low, b_mid)
        else:
            return median_helper(A, B, a_low, a_mid, b_mid, b_high)
    else:
        if A[a_mid] < B[b_mid]:
            return median_helper(A, B, a_low + 1, a_high, b_low, b_high - 1)
        else:
            return median_helper(A, B, a_low, a_high - 1, b_low + 1, b_high)

def median(A, B):
    assert len(A) == len(B)
    return median_helper(A, B, 0, len(A) - 1, 0, len(A) - 1)
```

- (b) We prove that the following invariant holds each time we call median helper:

Invariant: If the length of both subarrays is k (i.e. $a_{high} - a_{low} + 1 = b_{high} - b_{low} + 1 = k$), then the k th smallest element contained in the subarrays $A[a_{low}..a_{high}]$, $B[b_{low}..b_{high}]$ is the median of the $2n$ elements contained in the larger arrays A , B .

Initialization: At the start of the algorithm, we have $a_{low} = b_{low} = 0$ and $a_{high} = b_{high} = n - 1$. So $k = n$, the subarrays are the same as the larger arrays, and the k th smallest element in the two subarrays is the n th smallest element in the two larger arrays.

Maintenance: Let $a_{mid} = \lfloor (a_{low} + a_{high})/2 \rfloor$ and $b_{mid} = \lfloor (b_{low} + b_{high})/2 \rfloor$. Here we have two cases.

Case 1: k is odd. If $A[a_{mid}] < B[b_{mid}]$, then there are at least $(k-1)/2 + (k-1)/2 + 1 = k$ elements greater than a_{mid} , since there are at least $(k-1)/2$ elements in A that

are greater than $A[a_{mid}]$, and there are at least $(k - 1)/2$ elements in B that are greater than $B[b_{mid}]$ (and therefore $A[a_{mid}]$), and then there is the element $B[b_{mid}]$ itself. Therefore, all elements in A that are less than $A[a_{mid}]$ must be less than the median (since they must be in the first $k - 1$ elements). Similarly, there are at least k elements less than $B[b_{mid}]$, so all elements in B that are greater than $B[b_{mid}]$ must be greater than the median.

If we throw out the $(k - 1)/2$ elements in A less than $A[a_{mid}]$ and the $(k - 1)/2$ elements in B greater than $B[b_{mid}]$, we are discarding an equal number of elements above and below the median, so the subarrays of A and B are still the same length as each other, and the median of the new arrays must be the median of the old arrays. This maintains the invariant.

If $A[a_{mid}] > B[b_{mid}]$, the argument is exactly the same.

Case 2: k is even. If k is even, we try to reduce it to a case where k is odd. Here there are $k/2$ elements above $A[a_{mid}]$ in A , and $k/2 - 1$ elements below $A[a_{mid}]$ in A (similarly for B).

Suppose $A[a_{mid}] < B[b_{mid}]$. Then we want to discard $A[a_{low}]$ and $B[b_{high}]$. There are at least $k + 1$ elements greater than $A[a_{low}]$ (namely, all the other entries of A , plus $B[b_{mid}..b_{high}]$ which is guaranteed to contain at least two elements). Furthermore, there are at least k elements smaller than $B[b_{high}]$. Therefore, $A[a_{low}]$ must be below the median and $B[b_{high}]$ must be above the median, so we can discard them, maintaining the invariant, and moving back to a case where k is odd.

Termination: The algorithm always recurses on smaller and smaller subarrays, and it will eventually reach a state where $k = 1$. This is when it terminates. In this case, the smallest of the two elements contained in the subarrays is the median of the $2n$ elements, and the algorithm returns that element.

(c) Let $Q(n)$ be the running time of the algorithm. We have the recurrence

$$Q(n) = \begin{cases} 2 & \text{if } n = 1 \\ Q(\lceil n/2 \rceil) + 2 & \text{if } n \text{ is odd} \\ Q(n - 1) + 2 & \text{if } n \text{ is even} \end{cases}$$

We show that $Q(n) \leq c \log n$ for some constant c , as long as $n \geq n_0$ for some n_0 . We will pick $c = 4$ and $n_0 = 2$, and we will see below why we chose these values.

As our inductive hypothesis, let $Q(k) \leq c \log k$ when $k < n$. Then we break our induction into two cases.

Case 1: n is odd. Then

$$\begin{aligned} Q(n) &\leq Q(\lceil n/2 \rceil) + 2 \\ &\leq c \log(n/2 + 1/2) + 2 \\ &\leq c \log(3n/4) + 2 \quad (\text{for } n \geq 2) \\ &= c \log n - c \log(4/3) + 2 \\ &\leq c \log n \quad (\text{when } c > 2/\log(4/3)) \end{aligned}$$

Case 2: n is even. Then

$$\begin{aligned}
 Q(n) &\leq Q(n-1) + 2 \\
 &\leq Q(n/2) + 4 \\
 &\leq c \log(n/2) + 4 \\
 &= c \log n - c + 4 \\
 &\leq c \log n \quad (\text{when } c \geq 4)
 \end{aligned}$$

So the induction is proven when $c \geq 4$ (which is the most stringent of the requirements on c above) and when $n_0 = 2$.

3. Suppose you want to sort an array A of n numbers (not necessarily distinct), and you are guaranteed that all the numbers in the array are in the set $\{1, \dots, k\}$. A **“20-question sorting algorithm”** is any deterministic algorithm that asks a series of YES/NO questions (not necessarily 20 of them, that’s just a name) about A , and then *writes down* the elements of A in sorted order. (Specifically, the algorithm does not need to rearrange the elements of A , it can just write down the sorted numbers in a separate location).

Note that there are many YES/NO questions beyond just comparison-questions—for example, the following are also valid YES/NO questions: “If I ignored $A[3]$ and $A[17]$ would the array be sorted?” and “Did it rain today?”

- (a) (2 pts) Describe a 20-question sorting algorithm that will, for every input, ask only $O(k \log n)$ questions. Feel free to assume that the algorithm is also told n and k , although this isn’t necessary. [Hint: If you are stuck, first think about how you would do this with $\log n$ questions if $k = 2$. What would you need to know about the array to write down the sorted list of elements?] **[We are expecting a description of the algorithm and an informal (1-paragraph) argument that it achieves the desired runtime.]**
- (b) (2 pts) Prove that for *every* 20-question sorting algorithm, there is some array A consisting of n integers between 1 and k that will require $\Omega(k \log \frac{n}{k})$ questions, provided $k \leq n$. [Hints: Why is it sufficient for this problem to lower-bound the number of ordered arrays, instead of counting exactly? Once you have understood this, use a counting argument: how can you lower-bound the number of ordered arrays are there that consist of n integers $\{1, \dots, k\}$ (not necessarily distinct)? There are a number of ways to do this; we suggest you do NOT use Stirling’s approximation: you don’t need this in order to prove the result, and it will be complicated.] **[We are expecting a mathematically rigorous proof (which does NOT necessarily mean something long and tedious).]**

SOLUTION:

- (a) To output the sorted array, we need to know how many occurrences there are of each value $1, 2, \dots, k$. So for each of these we do a binary search on the number of occurrences: for example, we ask “Are there $n/2$ or more 1’s in the array?” and so on until we determine the number of occurrences for each of $1, 2, \dots, k$. Each binary

search takes $O(\log n)$ time and we do k searches, so in total this takes $O(k \log n)$ time.

Formally, the pseudocode is as follows.

```
def bsearch(value, low, high):
    if low >= high:
        return low
    mid = (low + high) / 2
    answer = ask("Are there more than 'mid' occurrences of 'value' in the array?")
    if answer is "yes":
        return bsearch(value, mid + 1, high)
    else:
        return bsearch(value, low, mid)

def main():
    counts = []
    for i = 1 to k:
        counts.append(bsearch(i, 0, n))
    for i, count in enumerate(counts):
        for j = 1 to count:
            print i
```

The above assumes knowledge of n and k (which is fine for the homework). If you did not know what these values were, you could also do a binary search for them. For example, you could ask “Is $n < c$?” for $c = 2, 2^2, 2^3, \dots$. If the answer is first “YES” for $c = 2^b$, then we binary search for n between 2^{b-1} and 2^b . This will take in total $O(\log n)$ time: $O(\log n)$ to find the range we need to binary search, and $O(\log n)$ to do the binary search. This same procedure can be used to find k .

- (b) An algorithm that asks at most c questions can have at most 2^c outputs because each sequence of answers can correspond to only one output. For any series of c yes/no questions, there are 2^c sequences of possible answers. So if there are N distinct sorted arrays (for given values of n and k) then any algorithm must ask at least $\log N$ questions in the worst case.

A lower bound on N , the number of distinct sorted arrays of length n containing the elements $\{1, \dots, k\}$, is given as follows: for each value $1, \dots, k-1$ let it occur anywhere between 1 and n/k times. Let the value k occur the number of times that will fill out the array to n numbers. This results in $\left(\frac{n}{k}\right)^{k-1}$ distinct sorted arrays. Thus we have a lower-bound on the worst-case number of questions: $\log \left(\frac{n}{k}\right)^{k-1} = (k-1) \log \frac{n}{k} = \Omega(k \log \frac{n}{k})$.

Alternative way to count N : the number of distinct sorted arrays is given by the number of ways to distribute n balls into k boxes, which is $\binom{n+k-1}{k-1} = \frac{(n+k-1)(n+k-2)\dots(n+1)}{(k-1)!} \geq \frac{n^{k-1}}{k^{k-1}} = \left(\frac{n}{k}\right)^{k-1}$.

4. Suppose that on your computer you have stored n password-protected files, each with a unique password. You’ve written down all of these n passwords, but you do not know

which password unlocks which file. You've put these files into an array F and their passwords into an array P in an arbitrary order (so $P[i]$ does not necessarily unlock $F[i]$). If you test password $P[i]$ on file $F[j]$, one of three things will happen:

- 1) $P[i]$ unlocks $F[j]$
- 2) The computer tells you that $P[i]$ is lexicographically smaller than $F[j]$'s true password
- 3) The computer tells you that $P[i]$ is lexicographically greater than $F[j]$'s true password

You **cannot** test whether a password is lexicographically smaller or greater than another password, and you **cannot** test whether a file's password is lexicographically smaller or greater than another file's password.

- (a) (3pts) Design an randomized algorithm to match each file to its password, which runs in expected runtime $O(n \log(n))$. **[We are expecting: pseudocode and/or an English description of an algorithm.]**
- (b) (2pts) Explain why your algorithm is correct. **[We are expecting: an informal argument (a paragraph or so) about why your algorithm is correct, which is enough to convince the reader/grader. You may also submit a formal proof if you prefer.]**
- (c) (2pts) Analyze the running time of your algorithm, and show that it runs in expected runtime $O(n \log(n))$. **[We are expecting: a formal analysis of the runtime.]**

SOLUTION.

- (a) We first give an english description and then the pseudocode. In English, the algorithm is very much like quicksort. For the pivot, we pick a random file $F[i]$. Now, for the partition step, we test $F[i]$ against each $P[j]$. For those $P[j]$ which are lexicographically less than $F[i]$'s password, we put them in an array P_{less} , and for those lexicographically greater, we put them in an array $P_{greater}$. We will also eventually find $F[i]$'s correct password, say it's $P[i^*]$, and we will label this P_{match} .

Next, we do the same thing to the files, using $P[i^*]$ to partition them: If $F[j]$'s password is lexicographically less than $P[i^*]$, we will put it in F_{less} , and so on.

Then we match $F[i]$ to $P[i^*]$, and recurse on our arrays (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$. We stop when the arrays are empty.

The following pseudocode captures the idea described above and meets the desired runtime but does not perform in-place pivoting; it is space inefficient.

```
def unlock_files(F, P):
    Let n = length of F (which equals length of P)
    if n == 0:
        return
    Let i = random integer from 1 to n
    Let P_less = [], P_greater = [], P_match = null
    for j = start to end:
```



```

    result = test_file(F[i], P[j])
    if result == 'unlock':
        P_match = P[j]
    else if result == 'P[j] is smaller':
        P_less.append(P[j])
    else if result == 'P[j] is greater':
        P_greater.append(P[j])
Let F_less = [], F_greater = []
for j = start to end, skipping i:
    result = test_file(F[j], P_match)
    if result == 'P_match is smaller':
        F_greater.append(F[j])
    else if result == 'P_match is greater':
        F_less.append(F[j])
register_match(F[i], P_match)
unlock_files(F_less, P_less)
unlock_files(F_greater, P_greater)

def main():
    unlock_files(F, P)

```

- (b) We verify correctness by induction.

Inductive hypothesis: We use the inductive hypothesis that at the return of each recursive call to our algorithm, the files in F are matched to the passwords in P for arrays of size less than n .

Initialization: Empty arrays are already matched, so the base case holds trivially.

Maintenance: For arrays of size n , $P[i^*]$ matches $F[i]$. We recurse on (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$. F_{less} has size less than n and consists of all files in F that have passwords lexicographically smaller than $F[i]$, and P_{less} contains all passwords in P that are lexicographically smaller than $P[i^*]$, and so there is a matching between the files in F_{less} and the passwords in P_{less} . The other side follows similar logic. Both the subproblems (F_{less}, P_{less}) and $(F_{greater}, P_{greater})$ have lengths less than n , so by the inductive hypothesis, our algorithm matches those files and passwords. Finally, the algorithm matches $F[i]$ with $P[i^*]$, so all n files and passwords are matched.

Conclusion: By induction, the inductive hypothesis holds for all n , hence the files in F are correctly matched to the passwords in P . Thus our algorithm is correct.

- (c) To analyze this, we may do an analysis similar to the one we did in class for Quick-Sort. We outline the approach here. Let P^* denote the array of passwords in lexicographic order, and let F^* denote the files sorted to match. Let $X_{i,j}$ be an indicator random variable which is equal to 1 if file $F^*[i]$ is ever compared against password $P^*[j]$, and 0 otherwise. Because these comparisons dominate the runtime (and because each pair $F^*[i]$ and $P^*[j]$ is compared at most once), it suffices to count them to get a $O()$ bound on the runtime.

Thus, the expected running time is $O(E[\text{number of comparisons}])$, which is

$$E \left[\sum_{i \neq j} X_{i,j} \right] = \sum_{i < j} EX_{i,j} + \sum_{j < i} EX_{i,j}.$$

First, focus on $EX_{i,j}$ for $i < j$. Just as in class, $EX_{i,j}$ is the probability that $F^*[i]$ and $P^*[j]$ are ever compared. This happens if and only if $F[i]$ is picked as a pivot first out of all of the files $F^*[i], F^*[i+1], \dots, F^*[j]$. Indeed, if some other file $F^*[k]$ for $i < k \leq j$ in that range were picked first, then $P^*[j]$ would be moved into `P_greater`, since it is lexicographically larger than $P^*[k]$ which would be chosen as `P_match`; $F^*[i]$ would be moved into `F_less` for the same reason. But then $F^*[i]$ and $P^*[j]$ would end up in separate recursive calls and would never be compared. Thus, for $i < j$,

$$EX_{i,j} = \frac{1}{j - i + 1},$$

because there is only one choice ($P^*[i]$) out of all $j - i + 1$ possible pivot choices that would cause $P^*[j]$ and $F^*[i]$ to be compared. Similarly, for $i > j$, we have

$$EX_{i,j} = \frac{1}{i - j + 1}.$$

Just as in class, when we add these up, we see that the expected running time is $O(n \log(n))$.

5. (7pts, 1 pt per part) In the `select` algorithm from class, in order to find a pivot, we break up our array into blocks of length 5. Why 5? In this question, we explore `select-3`, in which we break up our array into blocks of length 3. As you go through this problem, feel free to refer to the course notes from 4/12 and the week 3 recitation notes. In addition, to simplify your logic, you should assume throughout this problem that your array is a power of 3.

- (a) Consider the pseudocode for `select` and `choosePivot` on pages 2 and 4 of the April 12th course notes. In this pseudocode, we break up our array into blocks of size 5. What change(s) would you need to make to this pseudocode in order to write `select-3` and `choosePivot-3`? **[We are expecting a one or two sentence description of changes made, so that a 106b student would know exactly what to do.]**

SOLUTION. On the first line of `ChoosePivot-3`, you would need to split A into $g = \lceil n/3 \rceil$ groups instead of $g = \lceil n/5 \rceil$.

- (b) Prove that the recursive call to `select-3` inside of `choosePivot-3` is on an array of size at most $n/3$. You should assume that your array is a power of 3. **[We are expecting a rigorous proof.]**

SOLUTION. In `choosePivot-3`, we break up our array into blocks of size 3. Because our array is of size power of 3, we know that this will result in exactly $n/3$ groups. Each group has just one median, so our recursive call will be on $n/3$ values.

- (c) Prove that the recursive call inside of **select-3** is on an array of size at most $2n/3+2$. You should again assume that your array is a power of 3. (Hint: it might be helpful to note that $\lceil x \rceil \leq x + 1$.) **[We are expecting a rigorous proof.]**

SOLUTION. We know that our median, p , is the median of $p_1, \dots, p_g$ (where g is the number of distinct groups we broke our array into, namely $g = n/3$). Then there are at least $\lceil g/2 \rceil - 1$ group medians which are *smaller* than p . Each median p_i has two other elements in its group, one smaller and one larger, which we will call s_i and l_i respectively. But if $p > p_i$ for some group median p_i , then $p > s_i$ also. But also, p itself has a smaller element within its group. Taking all this into account, we can see that the number of elements less than p is at least $2 \cdot (\lceil g/2 \rceil - 1) + 1$. Then the number of elements which can be *bigger* than p is at most

$$\begin{aligned} (n-1) - (2 \cdot (\lceil g/2 \rceil - 1) + 1) \\ &= n - 2\lceil g/2 \rceil \\ &\leq n - 2(g/2) \\ &= n - g \\ &\leq n - n/3 \\ &= 2n/3 \end{aligned}$$

Thus we have shown that the number of elements that can be bigger than p is at most $2n/3$. Because the problem is completely symmetric, we also know that the number of elements that can be smaller than p is at most $2n/3$, also.

This tells us that the recursive call inside of **select-3** can be on an array of at most size $2n/3$, as desired, and we are done.

- (d) Explain why the work done within a single call to **select-3** and **choosePivot-3** on an array of size n is $\Theta(n)$. **[We are expecting a few sentences of explanation. You do *not* need to do a formal proof with the definition of Θ .]**

SOLUTION. Within a single call, we do almost exactly the same amount of work as in the version of the algorithm taught in class; the only difference is the size of the blocks. The work we do is

- split our array into groups of size 3 and find the median of each group, which is $\Theta(n)$ because there are $\Theta(n)$ groups and within each only a constant number of elements
- once we know the median of medians, p , we partition around p . This takes $\Theta(n)$, since we only have to do one compare and one move/write for each element.
- once we have partitioned, we figure out which side to recurse on and what index we're looking for on that side. This is just an arithmetic problem, and actually takes $O(1)$.

All together, this is $\Theta(n)$ work.

- (e) Using what you proved in (b), (c), and (d), write down a recurrence relation for the runtime of **select-3**. (You can do this problem even if you did not complete all of

(b), (c), and (d).) [We are expecting a one-line answer with your recurrence relation.]

SOLUTION. $T(n) = T(n/3) + T(2n/3) + \Theta(n)$

- (f) Is select-3 $O(n)$? Justify your answer. [We are not expecting a formal proof, but you should describe clear reasoning, such as analyzing a tree, unraveling the recurrence relation to get a summation, or attempting (and either succeeding or failing) at the substitution method.]

SOLUTION. No, it is not $O(n)$. We will imagine drawing a tree.

At the top level of our tree, we have only one problem of size n , and we do $\Theta(n)$ work. Then, at the next level, we do $\Theta(n/3) + \Theta(2n/3) = \Theta(n)$ work. In fact, if we look at the two children of a particular node, we notice that the amount of work done within each of those two children is exactly equal to the amount of work done in the parent! This is true all the way down the tree, so we can see that there is $\Theta(n)$ work done at every level. Moreover, since we either multiply the problem by $1/3$ or by $2/3$ each time, we can see that there are $\Theta(\log n)$ levels. (Depending what route you take through the tree, it could range from $\log_3(n) + 1$ at shortest to $\log_{3/2}(n) + 1$ at longest, but the difference between $\log_3(x)$ and $\log(3/2)(x)$ is just a multiplicative constant, which we ignore in the $\Theta()$ notation. Then overall we can see that the runtime is $\Theta(n \log n)$, so it is not $O(n)$.